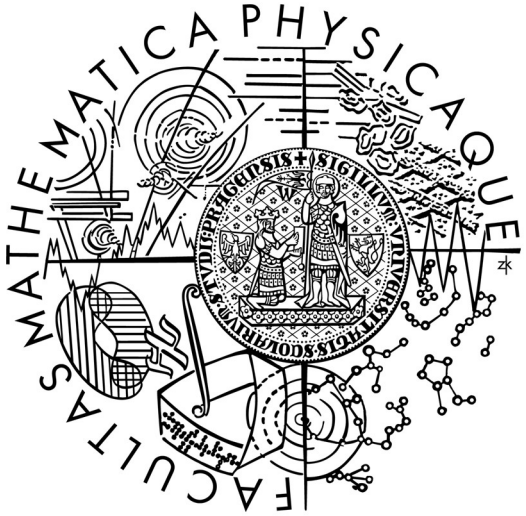
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Josef Reidinger

Syntax-based extraction of component behavior specifications

Department of Software Engineering
Advisor: RNDr. Tomáš Poch
Study Program: Computer Science, Software Systems

I would like to thank my advisor for his valuable comments and suggestions. I also want to thank Jan Kofron for his help with the communication with developers of SAMM. I also want thank Veronika Houdkova for checking the English grammar. Finally I want to thank my family for their support.

This work was funded in the context of the Q-ImPrESS research project (http://www.q-impress.eu) by the European Union under the ICT priority of the 7th Research Framework Programme.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with lending of this thesis.

V Praze dne *10.12.2009* *Josef Reidinger*

Content

Název práce: *Syntax-based extraction of component behavior specifications*

Autor: *Josef Reidinger*

Katedra (ústav): *Katedra softwarového inženýrství*

Vedoucí diplomové práce: *RNDr. Tomáš Poch*

e-mail vedoucího: *poch@dsrg.mff.cuni.cz*

Abstrakt:

Pro velké softwarové systémy je možné použít komponentový způsob vývoje. Jeho výhody jsou, že lze snadno části použít opakovaně a pokud obsahuje i popis chování, tak lze systém snadněji analyzovat. Komponenty jsou tvořeny popisem rozhraní a volitelně popisem chovaní. Popis chování je možné zkoumat na porušení omezení jiných komponent při jejich použití, jestli chování je očekávané a jestli komponenty k sobě pasují. Popis chování definuje omezení použití komponenty a jak reaguje na zavolaní každé své poskytované metody. Reakce popisuje pořadí a četnost volání metod požadovaných komponent. Manuální definování chovaní komponenty je náchylné k chybám a mělo by se zautomatizovat.

Výsledkem této práce je nástroj který z kódu, který splňuje omezení, je schopen vygenerovat automaticky popis chování. Nástroj je možné použít pro automatické generování popisu chování při převodu běžného softwaru na komponenty nebo při úpravě komponenty ke kontrole změny chování změnené komonenty. Práce také obsahuje prozkoumání možných technologií využitelných pro analýzu. Nástroj je součástí mezinárodního Q-Impress projektu a využívá jeho nástroje pro svoji funkci a naopak.

Klíčová slova: *Behavior protokoly, Analýza chovaní, Q-Impress, Q-Abstractor*

Title: *Syntax-based extraction of component behavior specifications*

Author: *Josef Reidinger*

Department: *Department of Software Engineering*

Supervisor: *RNDr. Tomáš Poch*

Supervisor's e-mail address: *poch@dsrg.mff.cuni.cz*

Abstract:

There is a component based paradigm which can be used for large software systems. It has advantage that its parts - components can be resused and if contain behavior description it allows many behavior analyses. A component contains static description of interface for reusing and optionally behavior description for correctness analyses. It is possible to further analyze behavior description whether a component does not break restrictions of the required components, whether the resulted behavior is the expected one and whether all components fit together. Behavior description defines restrictions on component methods and how it reacts to an invocation of method. The reactions specify order and workflow of invoking methods from the required interfaces. Manual writing of component behavior is prone to bugs and it should be automated.

The result of this work is a tool which can generate automatic behavior description from the source code that does not violate restrictions. The tool can be used to add behavior description during transformation legacy systems to component based ones or to check how source code changes affect behavior description. The work also contains an evaluation of possible technologies which can be used for analysis. The tool is a part of the international Q-Impress project and uses other project tools and, in return, it is used in other project tools.

Keywords: *behavior protocols, behavior analyzer, Q-Impress*

1 Introduction

Nowadays software is getting more complex and difficult to develop. Software systems often contain certain functionalities, e.g. database access, creating and handling graphical user interface or parallel execution of different tasks. Creating these parts for each system all over again is time consuming and hard to maintain. If these parts are identified, separated and reused it decreases the time to develop them and it also decreases the number of bugs in these parts.

The first way that comes to your mind is to move the functionality to libraries. However, they have the same problem with large codes as monolithic software systems. A code contains certain parts that are not on the level of abstraction which is expected from a library, such as handling optimal pixel rendering strategy for a library which provides window GUI. A library also usually needs a functionality which is not its main goal e.g. Graphical User Interface library for a window system needs a thread library for a particular separated rendering thread and a low level rendering library to draw its elements. This usually leads to the decision which implementation of a required functionality to use. The author of the software should choose a library and stick with it. A better option is when for a certain functionality there is a defined interface and the implementation of the interface can be changed if required, e.g. when an implementation becomes deprecated or when one needs to use their software on another platform.

Another problem of libraries is that functions and data structures in these libraries often have some restrictions on use e.g. correct initialization of a thread before use or flushing a buffer after the end of your work. The restriction is often only documented and lets the developer to use it correctly in a code. A better way is to use runtime asserts, but it shows incorrect uses only after a program starts. Automatic checking of restrictions provides quick response for the developer and eases development of large systems which have expensive testing.

Separated parts is also good indication that this part could be run on separate system and benefit from distributed computing which nowadays became more important. A libraries does not support it generally.

Component based development paradigm tries to solve these problems. Components are independent parts of a software system. Each component has some interfaces attached and it provides them itself. A software system communicates only via interfaces so component requirements are explicitly specified. The components that provide the requested interfaces can be easily interchanged. Remaining problem is missing restriction on used interfaces or component.

A component or an interface can also have its behavior protocol attached. A behavior protocol among others defines usage restrictions e.g. restriction to run initialization before other methods or

to call finalization only once. Behavior protocols are high level specifications and they capture the finite sequences of method calls allowed on the component [1]. If each component has its Behavior protocol, then an incorrect use of the component can be detected by composition of protocols at final composited system. More information can be found in the chapter about behavior protocols (Behavior Protocols Family).

1.1 Q-ImPrESS project

This thesis is a part of the Q-ImPrESS project. The project is focused research project funded by the European Union. Several European universities and companies participate on the project. The project aims to bring component[1] orientation to critical applications such as communication, production controlling and enterprise application, all of which are oriented to provide stable product.

The main aim of the project is to provide methods for quality-driven development and evolution. The methods from quality-driven development are used from the first application design to the final deployment and maintenance. It aims also to create models from legacy applications and automatic generation of model behavior from implementation helps create better project.

1.2 Goals

The goal of this thesis is to implement a tool (Q-Abstractor) capable of extracting high level behavior specification from the primitive component implementation provided in the form of Java sources. The tool will be modular to allow various extraction strategies and it will also exploit third party tools providing information that is useful to obtain more precise and accurate result. The expected result specification allows formal verification of component system in context of Q-Impress workflow. This work should be used as a part of a tool for automatic checking of the correctness of components communication and for checking if the implemented code matches the expected application behavior.

1.3 Structure of the Text

This work is divided into nine chapters. The first chapter is an introduction which presents the motivation and initial knowledge to a reader with general programming skills. The second chapter provides the theoretical background for this work and the used techniques. It provides specific knowledge which is needed to understand later chapters and it extends the information from the first chapter which only contains a rough overview.

The third chapter describes how this work achieve its goal without any implementation details.

1 Q-Impress project uses the word 'service' instead of 'component'

It only provides a description of how to solve that problem.

The fourth chapter describes the techniques and technologies which are considered to be used in implementation. Since some technologies were found as too complex or too generic for this problem, this chapter also contains a discussion about technologies and their usability for this work.

The fifth chapter contains implementation details and limitations of the current implementation and the sixth chapter contains a few ideas how to extend functionality. The seventh one makes comparison of this work and other works on related to this topic. The eight chapter is the conclusion of this work.

The appendix A contains installation and user guides. It also contains instructions how to run examples attached to this work. The appendix B contains content of included CD.

2 Background

2.1 Component models

As already mentioned in the introduction, component systems encourage the reuse of a code across different applications. There are many component systems which could be used [2]. A component system describes how to define interfaces and components and how they are joined together. Component systems range from simple models to more complex ones, featuring composite components, contingency or various connection restrictions.

A minimal component system contains only components which have their interfaces. The interface is implemented directly by underlaying language. Each component must specify which interface provides. The provided functionality is represented by methods and is called provided methods. The minimal component system loads components on demand during run. Thus, all errors are discovered only during execution. Examples of such simple systems are Microsoft Component Object Model (COM) or Linux's Desktop Bus (DBus). The standard interface (IUnknown for COM and Introspect for DBUS) which must provide each component has a method that says what another interfaces are provided by this component. Both example systems also have a predefined service which has the information about which component provides which interface (Registry for COM and a few DBus daemons for DBUS (session, system and system specific)). Therefore, the service informs the components which component provides the requested interface.

A typical component model defines required interfaces, in addition. A required interface is required functionality, which the component needs to function. An explicitly required interface has the advantage that the component system could be unify together before run since it knows all the required components. The examples of a flat component model are Corba component system or Sun Microsystems Java Enterprise Beans.

Hierarchical component model introduce the concept of primitive components and composite components. Primitive components are similar to components in the typical component model. The whole primitive component is implemented in the underlaying language. A composite component is composed of another components called subcomponents. A composite component could enclose functionality of low level components and act as a high-level component which provides high-level methods and requires a union of the required methods of the subcomponents. Frame and architecture are used for a precise description of the composite component.

Frame is a black-box view of a component and is similar to simple component's description. It defines provided and required interfaces for each component. Each interface has its type that allows

easier sharing of the same interfaces between frames. The sharing is done only by name reference to a type. Architecture is a grey-box view of a component. Architecture specifies what subcomponents are used in the component and specifies connections between subcomponents. There are several kinds of connection. One kind of connection joins the provided subcomponent method with composite component provided method, the other kind of connection joins the required interface of a subcomponent with the provided interface of another subcomponent. In the second case, the interfaces are identical. Unsatisfied required interfaces are added to composite component required interfaces. All primitive components have empty architecture, which indicates that the whole component is implemented directly in underlaying language. The provided interface of composite component does not have to be a union of its subcomponent's provided interfaces, some could be cloaked.

These complex models also allow attributes for an interface which specifies how the interface could be used and which usage restrictions must be respected. One of the common attributes of required interfaces is contingency which specifies how important an interface is for a component. If an interface is mandatory then it must be binded to another component. An optionally required interface does not have to be binded because the component could work without it. Database access is an example of a mandatory interface and logger or User Interface notification are examples of an optional interface.

An example of a more complex component model is SOFA2 which has a detailed description in [3].

2.2 Behavior Protocols Family

Behavior protocol is a unified way to capture the behavior of a component. It specifies its behavior as the record of incoming and outgoing method calls on the component's provided and required interfaces. Behavior protocol specifies how a component reacts after invoking a method from provided interface. The reaction is specified as calls of component's required methods.

Behavior protocols have three main cases of use which together provide a communication error analysis for the whole component system [4]. First is a Correctness check which checks if all required components are added and if all restrictions on usage of component provided methods are not broken. This check is invoked after the whole component system is put together. The second one is a substitutability check which checks if one component can be replaced by another one. The check can be used as a general check for two components or it can check substitutability for a specific component system where only a part of the component's interfaces are used. The third check is code conformance which checks if the real implementation matches the behavior

specification.

2.2.1 Behavior Protocols

Behavior protocols specify restrictions on provided methods and possible behavior after component method invocation. There is a grammar that allows to specify restrictions on provided methods, invoking outgoing calls and their order and quantity. It contains four basic events to model method calls - *?i.m^* (acceptance of invocation of method m on interface i), *?i.m$* (acceptance of method return), *!i.m^* (emission of method invocation) and *!i.m$* (emission of method return). These events form elementary protocols which can be further combined with regular operators to create more complex descriptions. Here are some regular operators: ';' (sequence), '+' (choice), '*'(repetition) and '|' (parallel composition – allows interleaving of events in operands). There are shortcuts for method events to ensure better readability. *!i.m* is the shortcut for *!i.m^; ?i.m$* and ? *i.m{...}* is the shortcut for ?*i.m^;...;!i.m$*. There exists the keyword *Null* for an empty protocol.

```
?Form.initialize {
1) !File.create; !dbAccess.open; !UI.createForm
};
(
  ?Form.edit {
2)  Null
  }+
  ?Form.clear {
3)  (!UI.clearForm*) +
4)  (!UI.destroyForm;!UI.createForm)
  }
5) )*;
(
  ?Form.destroy {
6)  (!File.delete | !dbAccess.close); !UI.destroyForm
  } +
  ?Form.save {
7)  ((!File.write; !File.sync) |
8)  (!dbAccess.store; !dbAccess.close) );
9)  !UI.closeForm;!UI.notification
  }
)
```

Figure 2 shows behavior protocol for a component which controls form and writes to file or database after saving a form.

Figure 2 shows example behavior protocol which saves result of form to file or database. The line marked 1 is initialization of component which initialize used components for file, database and form in defined order (sequence operator ;). The line marked 2 demonstrate is reaction on edit of form and the component does nothing with used components. The line marked 3 allows unlimited invocation of clearing form (operator *) or could be alternated by next line (operator +). On the line marked 4 is form destroyed and created again from scratch as alternative method from cleaning parts of form (brackets () ensures that whole sequence is alternate). The line marked 5 shows that cleaning or editing could be repeated (operator *) and then followed by following alternate methods (operator ; and brackets that enclose both methods). On the line marked 6 is reaction to destroy the form. It could delete file and close database concurrently (parallel operator |) and afterwards destroy UI representation of form (sequence operator ;). The line marked 7 shows that when form is saved then data is written to file and immediately synchronized afterwards. Data could be saved data to database concurrently on the line 8. The line marked 9 is form closed and shows notification about safe.

Behavior protocols are described in detail with the use of some outdated things at [5] and with a newer, less formal version specification at [6].

2.2.2 Threaded Behavior Protocols

Threaded behavior protocols (TBP) are extension of behavior protocols. The motivation for the extension of BP was the intention to make the specification language closer to imperative languages actually used to implement components. Thus, TBPs feature data (method parameters, state variables), threads and synchronization. Similarly to BPs, TBPs capture the traffic on the component's frame.

Types in TBPs can only be of enumerated value and there is a type declaration section that specifies the possible values for each type. State variables and method parameters use these types. While TBPs have their condition and repetition based on the value of a variable, BPs have only nondeterministic decisions. This means that TBPs use data to describe more precisely the behavior of a component. Since TBPs only use enumerated values, the data is reduced to simple values. However, the program work-flow can react to a certain range of values in the same way and so TBPs can reduce the whole range to one enumerable value. For example, if a program uses integer and it acts different if the integer is zero or non-zero, TBPs reduce the integer to enumerable type with values ZERO and NONZERO.

State variables can be assigned and used as conditions. Each variable has a defined type, name and initial value. One kind of type is mutex which is used as synchronization primitive between threads. All variables are only local for their component and cannot be accessed outside of the component.

TBPs have a provided method section which specifies valid usage of components provided methods. Dependencies between methods are specified in the same way as in Behavior protocols.

The behavior of provided methods is specified in following section. The section specifies what happens during execution of provided methods. Method body can contain invocations of method from required interface, assigning state variables, synchronization mutex variables, control flow statements with state variables in condition and return statement.

There is a thread section that specifies the set of threads. These threads are specified in the same way as behavior of provided methods. The set of threads is finite and a dynamic creation of threads is not allowed. Threads are sources of activity at component models. The threads call a method provided by another component and they switch to the context of the called component. After the method is finished, it switches back to the original component context. Threads can manipulate state variables of the component which is in the same context.

TBPs allow synchronized blocks which are enclosed in method call 'sync' with mutex variable as a parameter. Mutex has two states – locked and unlocked. Sync ensures that before executing the block the mutex variable is unlocked. It atomically locks this variable and it also ensures that this variable is unlocked again after the block is executed. This allows synchronization of critical sections between multiple threads.

Formal specification and detailed information can be found in [4]

2.3 Static analysis of code

By static code analysis we mean computer software analysis without execution of a program built from a code – simulating states of program are still counted as static analysis. Q-Abstractor analysis is also static so it is useful to know its limitation and benefits. Static analysis must know how software works inside and it tries to simulate or guess what the analyzed code does and how it reacts to outside stimulation. The alternative method is dynamic analysis which analyzes how a program interacts with the outside and it analyzes the resources used by the program. Both methods have their advantages and disadvantages. The advantages of static analysis are that it can analyze only a part of a program and that it can also analyze rarely executed parts of a code. The advantages of dynamic analysis are that it can measure precisely what resources a program needs to run and that the occurred behavior is the real behavior of the program without any bugs during simulation or

suggestion. The examples of static analysis are error checking during compilation or dead code elimination. The examples of dynamic analysis are profiling running application on system level by profiler or checking conditions by asserts in code.

One of the techniques of static analysis is points-to analysis. The main goal of points-to analysis is to decide whether two pointers point to the same physical data or not. Its secondary goal is to recognize the type of pointed data. Q-Abstractor use points-to analysis for inlining methods. It is quite often used in compilers for optimization in the final code. A compiler could optimize a variable which is only read, but if there are some aliases, then the compiler must decide if aliases point to the variable and cannot optimize. Points-to analysis is hard and it often cannot decide if pointers point to the same data or not. This limitation is often solved in the way that the result is only used if it is useful or otherwise a pessimistic variant is used.

Another technique is dead code elimination. Dead code is a code which cannot be reached by program work flow and is never executed. This code only confuses the reader and increases the size of program binary. Rigorous localization of each part of a code which is never executed is an almost impossible task in the currently used languages like Java. Identification of a class as a dead code is hard because even if it is not specifically allocated it can be dynamically allocated either through introspection or by the user. If dead code elimination is reduced to local statements, private variables and private fields of class, it then becomes quite an easy and an intuitive task and it is often used in static analyzer of code. Q-Abstractor do simplified dead code analysis.

There are another tasks for static analysis. Code style checker and quality of code measurer are often used for team cooperation. Security analyzer can be used for applications where security is important. These tasks is not used in Q-Abstractor and it use its benefits only during development.

3 Extraction of Behavior Protocols

3.1 Input/Output - SAMM Model

SAMM is the central meta-model for storing, analyzing and transforming component oriented architectures within the Q-Impress project. It contains a universal meta-model for sources named Generalized abstract syntax tree (G-AST) and it abstracts more programming languages and it therefore allows sharing a code for more source languages. There are analyzing tools and also tools like viewers or editors for this meta-model that allow easier development of other tools. SAMM allows the transformation from target component system to meta-model, which allows more general usage of this work. More information about SAMM can be found in [7]

3.1.1 High Level View

Q-Abstractor is integrated into set of Q-Impress tools. Q-Abstrator requires Generalized Abstract Syntax Tree (G-AST) for code analysis and component description. SiSSy analyze code and generate own syntax structure to database. Then G-AST Exported (part of SiSSy) takes the database and creates G-AST tree. Component description can be loaded from specification created by user or via SOMOX which fills prepared SAMM structure for component system description (Figure 3).

TBP specification produced by Q-Abstractor are used by another tools. The first, it provides the user top-level insight. The second, tool Badger can analyze TBP if component match together. The third, Java Path Finder (JPF) tool use also TBP. Figure 4 shows overview graph.

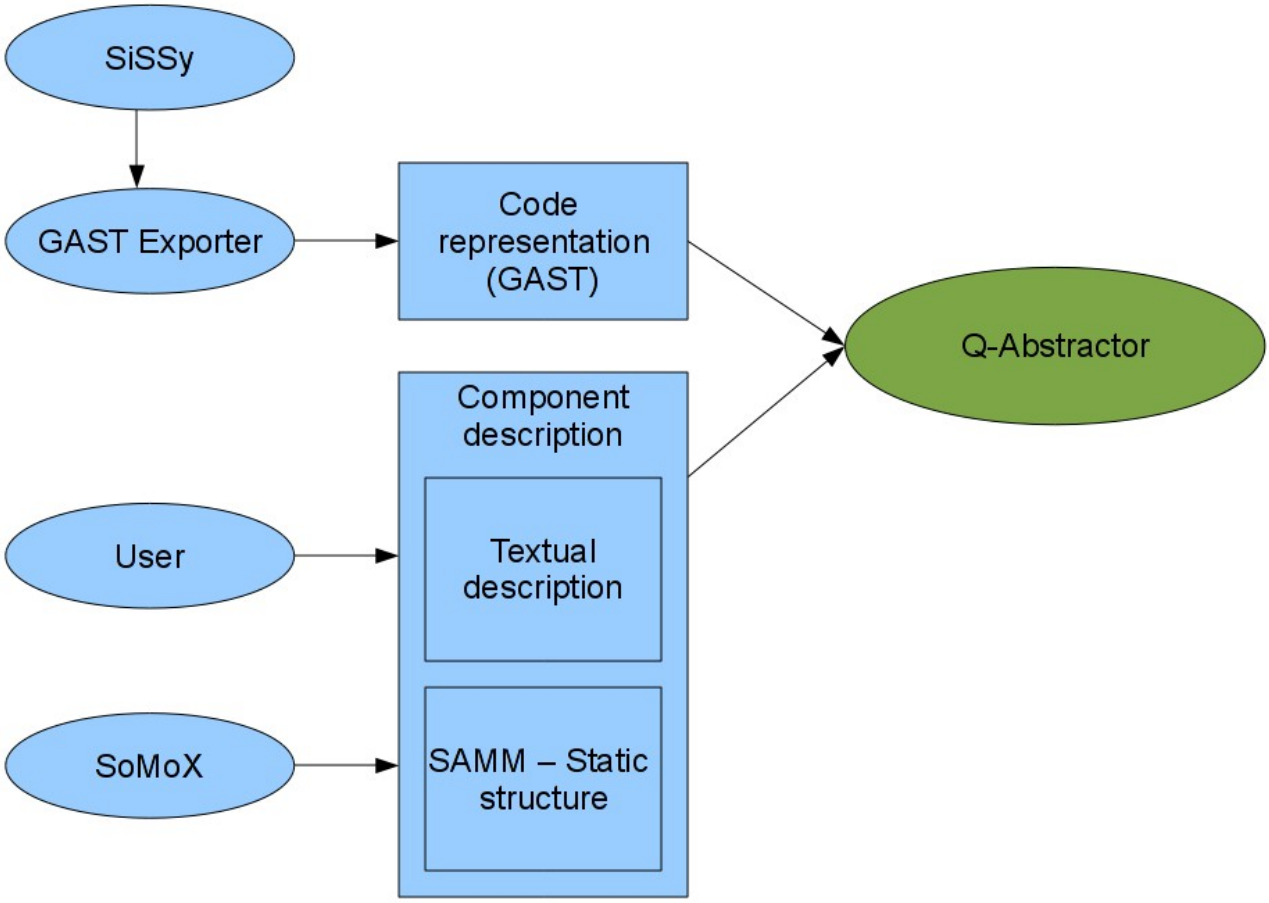


Figure 3 *shows tools used by Q-Abstractor. Elipse is tools and rectangles is produced data structures.*

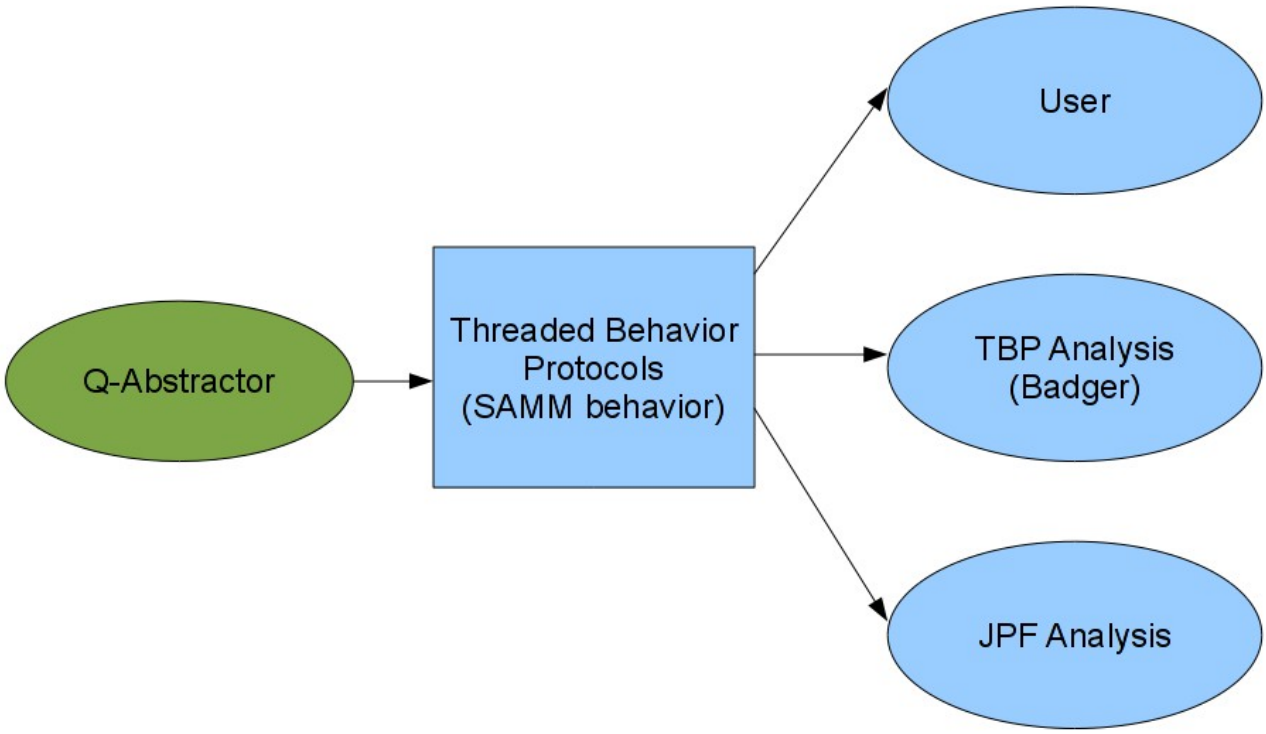


Figure 4 *shows tools which directly use output of Q-Abstractor.*

3.1.2 Relation of G-AST to Java/C

The G-AST model represents the whole program from packages and files to detailed statements in a tree structure. There is a root element on the top of the hierarchy. Packages lay under the root element and packages contain their classes. Classes contain methods and fields. Methods have statements. There are different kinds of statements. A simple statement represents a single line of code and it does not contain any structure, only expressions. A block statement contains sequence of

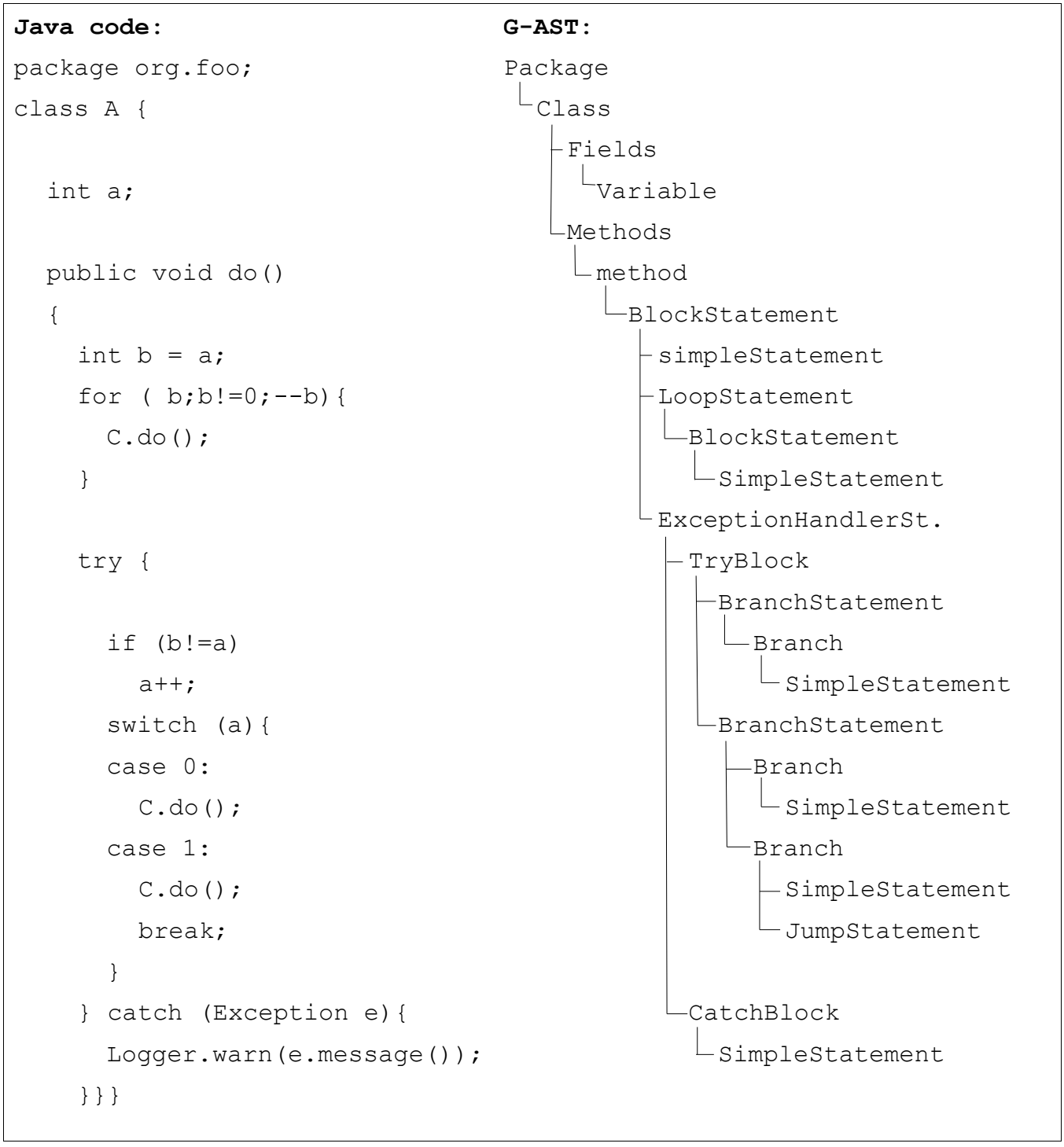


Figure 5 *Java code and its G-AST tree without expressions*

another statements – this block is represented by curly brackets in Java. A loop statement represents

any loop such as `do while`, `while` or `for` cycles and it contains one or more expressions and a body for cycle. The body is represented by a statement. A branch statement is used to model conditions in code such as `switch` or `if` statement. It contains expressions and a code which is executed in each branch. Exceptions are represented by an exception handler statement which contains a secured block, a finalized block and some catch blocks. A catch block contains a caught exception and a handling block. Example Java code and its G-AST tree without expressions (to simplify example) is at figure 5.

G-AST is created by SISSY (a tool developed by FZI – more at [8]) which takes classpath and fills its database with information about all classes and packages in the classpath. Afterwards the database is transformed into serialized form of G-AST. The serialized form has two separated xml files, one including statements without expressions and the second containing expressions. These two xml files are then merged into a single file and the single file is deserialized in the target program. So this is similar to the Unix way where each tool does one thing and their final sequence creates the expected result.

3.2 Transformation of G-AST to Behavior Protocol

The main part of this work is the transformation of G-AST input together with information about components to behavior protocol of the components. If G-AST input contains only "good" statements, then the program created by this work could map statements one to one to BP statements. That is exactly what the last transformation, which takes G-AST and transforms it by simple table rules (table 1) to behavior protocol, does.

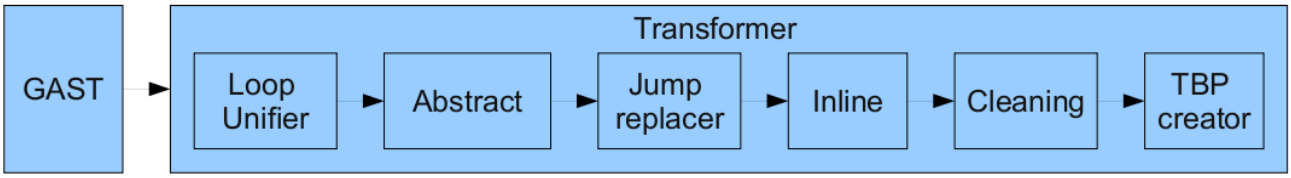


Figure 6 *transformation on G-AST tree*

Another transformations are needed to convert from the original G-AST to the G-AST without "bad" statements. What is a "bad" statement is in the description of each transformation as each transformation removes other problematic statements. Transformations are pipelined to the overall process, so each transformation expects that G-AST does not contain the statements removed by previous transformations. In other words, each step simplifies the Java code to correspond to a unified form required by subsequent step. The order of steps affects the precision of the final protocol because each transformation can lose some information, which can influence the precision of subsequent steps. All transformations except the final one produce a valid G-AST structure which simplifies testing and allows the use of the semi-results as the input to third party analysis tools. When transformation needs to replace an expression which cannot be represented in behavior protocol, it is replaced by `QAbst.nondet()` method which represents nondeterministic expression.

`a.do(); //required call`	`!A.do //where A is interface which provide method do`
`While (QAbst.nondet()) { … }`	`( … )*`
`If (QAbst.nondet()) { … }`	`( … )+Null;`

`Switch (QAbst.nondet()) {`
`case :`
`  …;`
`  break;`
`case:`
`  …;`
`  break;`
`}`
	`(`
	`  ( … )+`
	`  ( … )`
	`)`

`{ a.do(); b.do() }`	`( !A.do; B.do() )`

Table 1 *Java statements and its BP equivalent*

3.2.1 Loop Unify

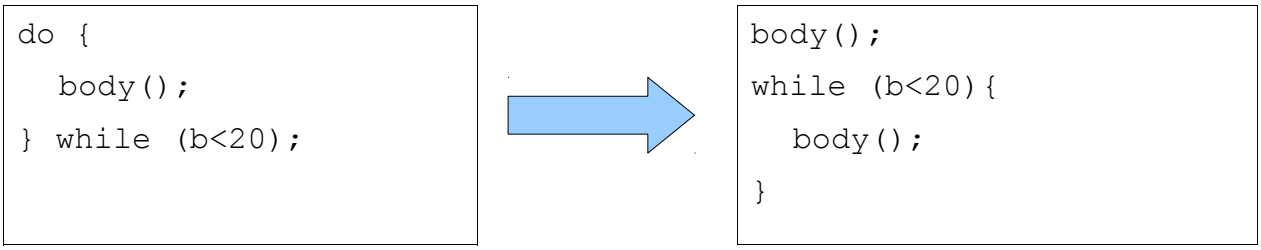
While BP supports a single repetition operator, Java features several kinds of loops. The number of expressions involved in loops as well as their meaning varies. Also if a loop condition contains a method call, the call is invoked at different time. The Java loop closest to the BP repetition operator is the `while` statement. Since all loop types can be transformed into another one, this transformation converts all loops to the `while` statement. The Loop Unify transformation is lossless. Because a lossless transformation does not affect the result precision and it makes it easier to handle expressions for the next transformation, it is used as the initial one.

Loop specific expressions `break` and `continue` which affect loop workflow are correctly removed by a following transformation – Jump replacer.

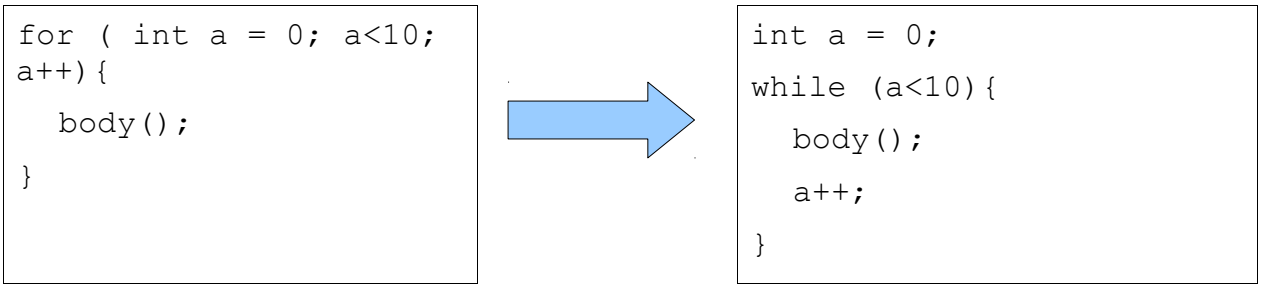
The transformation changes differently each type of loop:

While Loop - The statement is left without changes.

Do While Loop - The statement needs to duplicate its own body and put it before the resulting `while` statement.



For Loop - The initialization expression is put before the resulting `while` statement and the increment expression is appended to the end of the body. There is another obstacle related to the `continue` statement, since it needs to put the increment statement also before itself.



For Each Loop - The transformation for the statement is the least intuitive. A correct transformation requires transformation to `for` statement with the use of `Iterable` Java interface. There is a used trick that partially does the same as the next transformation – it moves an expression which gets objects (e.g. in *for(a : b.list())* it is the expression *b.list()*) before the loop. The resulting

loop with changed expression is then converted to `while` with `Qabst.nondet()`.

```
for ( Integer c :
integers ) {
   body();
}
```

```
integers;
while (QAbst.nondet()){
   Integer c;
   body();
}
```

3.2.2 Abstract

Target BPs do not contain any variables and they do not support any other expressions than simple method calls. The goal of this transformation is to split complex expressions into simple method calls and move expressions with side-effect outside of statements which use its result. Removing variables is not possible since they are still needed for methods calls to identify the variables on which the method is invoked.

The method calls transformed from expressions must respect the control-flow in those expressions (e.g. priority of operators, brackets). Moreover, Java features the short-circuit evaluation of boolean expressions. Such as the second operand of '&&' and '||' operators need not be used, so the method calls related to the second operand should be placed in the conditional statement.

The first goal is removing side-effects. Expression is moved outside of loop or branch statement and then apply on that expression same procedure as for other expressions. Expression from loop is also copied at the end of while loop to reflect repetition call of control expression.

```
while (a.do()){
   …;
}
if (a.do()){
   …;
}
```

```
a.do();
while(QAbst.nondet()){
   …;
   a.do();
}
a.do();
if (QAbst.nondet()){
   …;
}
```

The second goal is splitting expressions. Splitting expression is more complex operation. It traverses expression tree in depth and creates G-AST tree containing simple statements with simple expressions. If expression cannot affect resulting protocol, then it is removed. Different type of

expression is differently handled:

Constant and Variables - cannot affect result as it cannot contain any required call thus, it is removed.

Unary Operation - such a unary minus or not operator can affect result only if operand is important. It call simplify on operand and return its result without any modification.

Binary Operation without Short-circuit - is for example arithmetic operation (+,-,/,*). Both operands is evaluated so in this operation so result is block statement which contains result of simplified operands.

Binary Operation with short-circuit - is for example boolean operation (&&,||). Evaluation of the second operand depends on value of first operand. It must be properly reflected in result thus, the result of simplifying second operand is enclosed in condition evaluation. Result of simplifying the first operand together with enclosed result of second operand creates block statement which is result of simplifying binary operation.

```
a && b;
```

```
{
  a;
  if (QAbst.nondet())
    b;
}
```

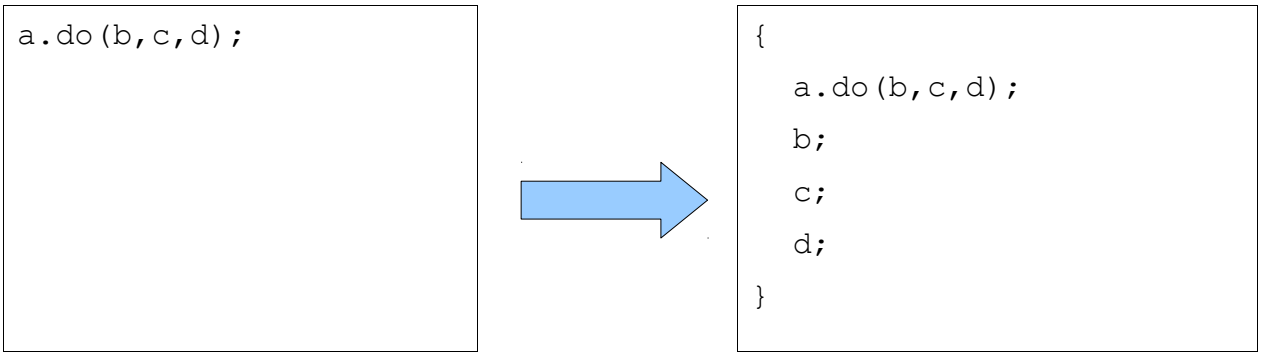
Conditional Evaluation - (operator ?:) creates block statement. It contains result of simplifying first operand because it is always executed. Result of simplifying second and third operand creates two branches in created branch statement.

```
a ? b : c;
```

```
{
  a;
  if (QAbst.nondet())
    b;
  else
    c;
}
```

Method Call - is important expression as it can be or lead to required method. Method call can have arguments which can also contain important expression. Thus, result of simplifying method call is block statement which contain method call itself and results of simplifying all arguments.

Because number of arguments is important to identify which method is invoked arguments is not removed and following transformation act as all method call doesn't contain anything important.



The transformation decreases the complexity of statements and expression handling, so remaining transformation could expect only expression which contain single method call. That is the main reason to use it as the second transformation in the stack.

3.2.3 Jump Replacer

Behavior protocols do not allow jumps like `return`, `break` or `continue`. Each jump can be removed if the rest of over-jumped statements is enclosed in conditional execution which depends on jump execution. There are no variables where the result of a jump could be stored, so all these conditions are nondeterministic. So in case of `break`, the jump stops the execution of the `switch` or loop statements, and so all remaining statements in the `switch` or loop statement must be enclosed in conditional execution. The statement `continue` only affects loops. The statement `return` affects the execution of the method body where it is used. Replacement of return statement is especially important for correct result of the following transformation 'inline' because if the transformation inlines the method body which includes `return` then the `return` jumps to a different place than it was intended.

The used G-AST implementation does not recognize the difference between `break` and `continue` and it also does not recognize the difference between `switch` and `if` statement, which makes it hard to correctly replace `break` and `continue` jumps. To replace it correctly, one needs to study the original code and match corresponding elements from the code to G-AST representation and replace it. Since it is out of scope of this work, `break` and `continue` jumps are not replaced, only logged and removed.

The transformation adds many conditional statements to the code and it can mess the final code. Some users can find it useless and for this reason the transformation is optional and can be disabled in configuration. Figure 8 shows example of that transformation.

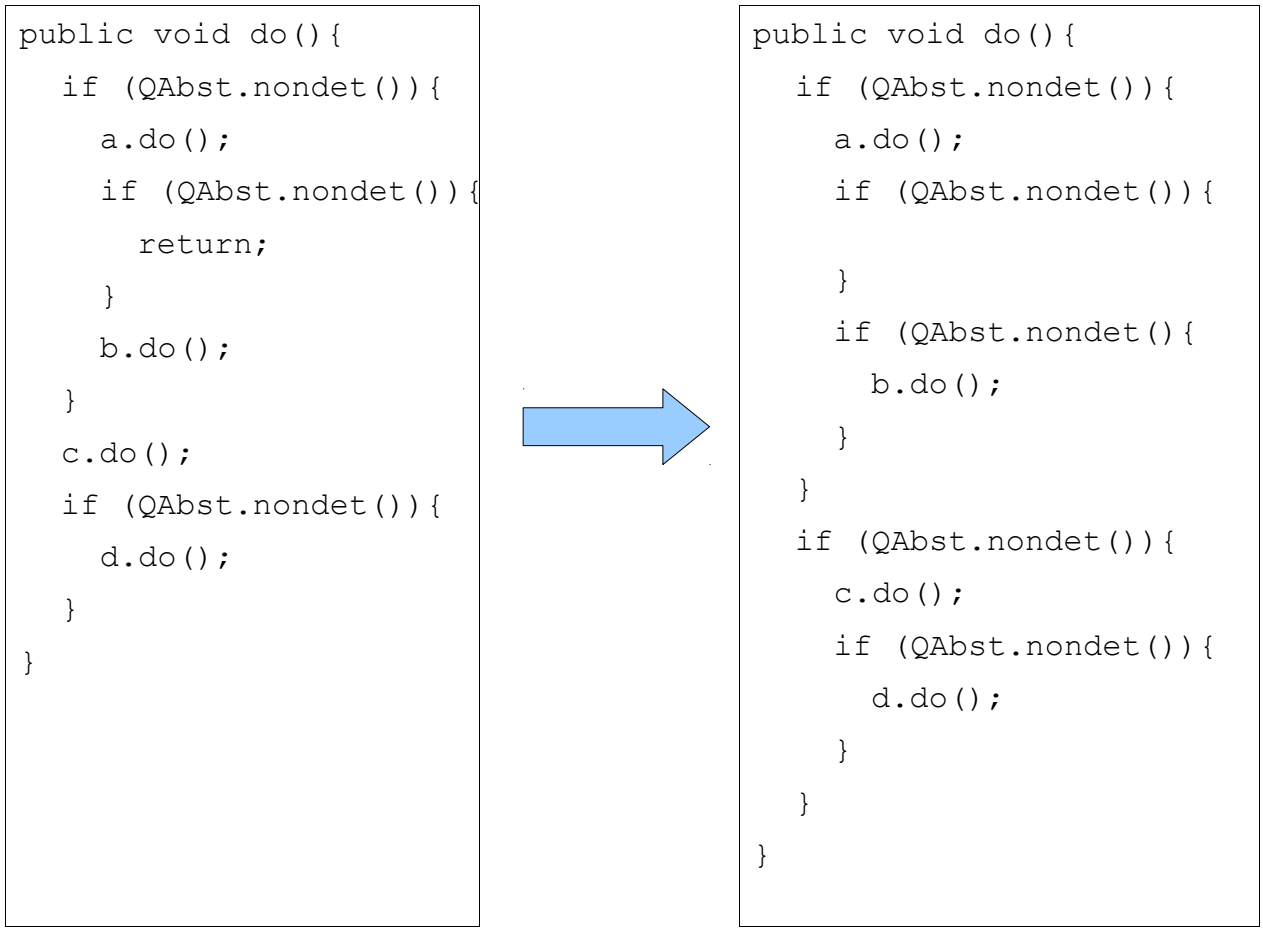


Figure 8 *example of jump replacer transformation*

3.2.4 Inline

Behavior protocols represent one component by one protocol. Component implementation can use more classes. It is an intuitive task to merge all classes to one which contains only provided methods which include only required calls. The implementation of a provided method does not have to call the required interface directly. The method call can go through many other methods of the component's internal objects before it reaches a method call on the required interfaces. As a result, the required method is called in response to the provided method invocation. Although such dependency is not directly apparent from the code, it must be reflected in the result. Merging is done directly via inlining bodies of the called methods to source methods which invoke the calls.

The transformation creates a new class with the same name as the component for easier identification. Provided methods from source classes are copied to the resulting class. Simple statements that do not contain a method call are removed. The evaluation of simple statements with a method call depends on the kind of method:

1. If the method is a required method, then the simple statement is kept in the result

2. If the method is outside of the component then simple statement is removed
3. Other methods are replaced by their bodies

Method replacement must be done recursively to ensure that the final methods contain only required method calls. Direct inlining restricts the transformation only to methods that do not contain recursive calls, even indirect calls via another method (e.g. method A calls method B which calls method C and C calls method A). If this case happens, then the transformation reports an error. A solution that removes the restriction exists but it is out of scope of this work. Filtering of methods outside of the component also adds limitations for callback functions. If a class outside of the component registers a required method as a callback function, it is not tracked and it will be missing in the final protocol.

The transformation also handles threads which are parts of a component. Method `run` from a thread is added with a special identification (for the following transformation) to the newly created class and the whole inlining process is run on this method.

Class inheritance adds two difficulties to inline process. The first difficulty is virtual method invocation. If instance of child class is assigned to a variable, then invocation method on the variable uses child's method instead of the method from the variable class. Assignments often depend on runtime decision so method body to inline cannot be unambiguously chosen. Resolving runtime type of variable is one of the goals of points-to analysis. The second difficulty is that a class on which a method is invoked need not contain the method. Any ancestor of the class could have this method implemented. Correct method implementation is the one which is included with the nearest ancestor in inheritance hierarchy. There are two procedures which try to resolve virtual methods.

Points-to analysis has same complexity as halting problem, because to correct and unambitious result exact run path is required. There is only approximately solution with set of potential classes. Replacement of method calls can result in more potential bodies so to obtain a correct result all possible method bodies are placed to nondeterministic switch. Simple solution of virtual methods returns all derived classes including the class itself. The solution however contains a lot of useless classes, and it produces big protocols for programs with large object hierarchy. The solution is always correct and never misses any possible code. More precise virtual method resolving is done via points-to analysis which returns a smaller set of potential variable runtime types. Figure 9 shows the final replacement part of 'inline' transformation, where at first runtime type of variable is inspected and then the transformation finds which method from inheritance hierarchy is called.

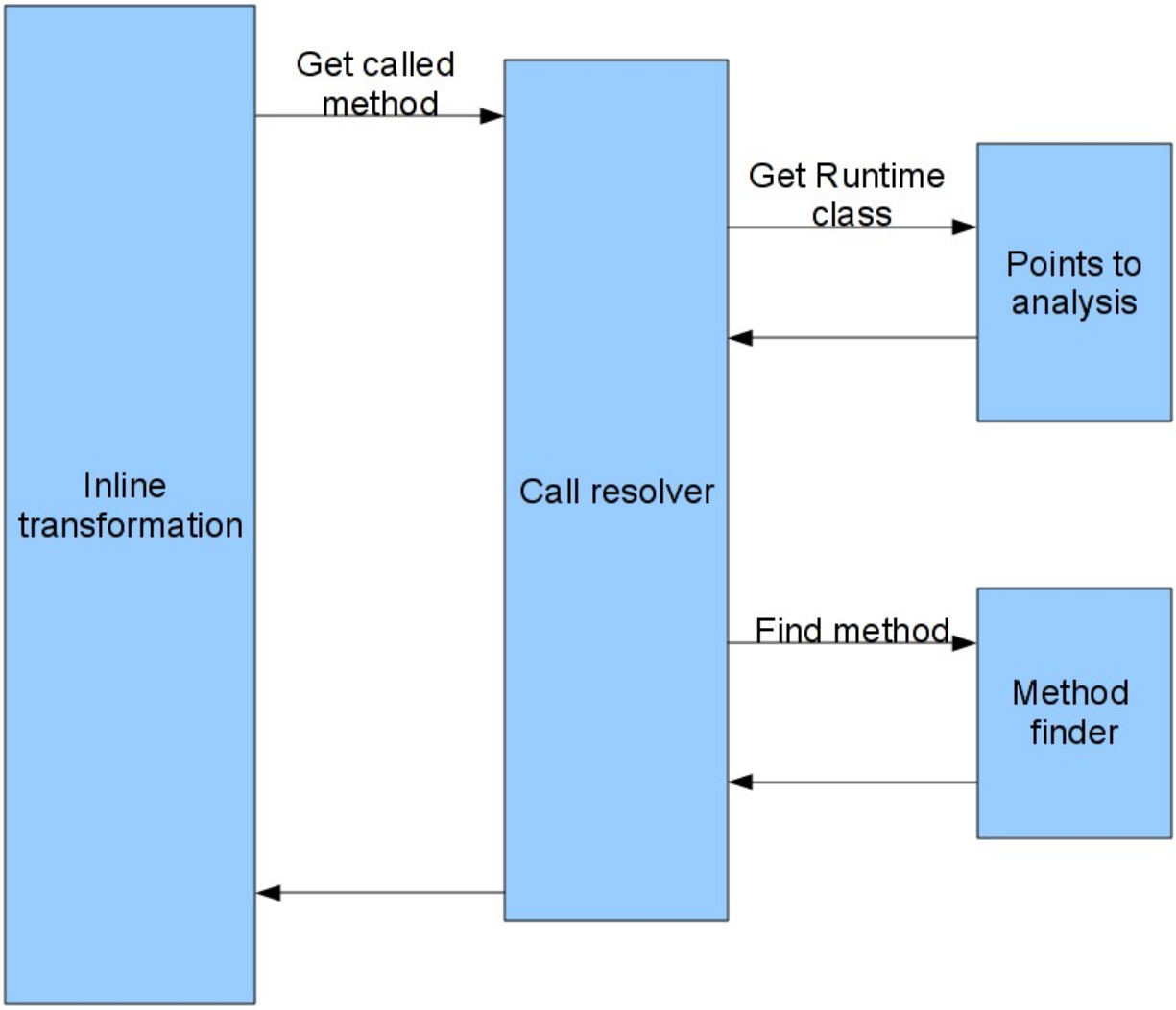


Figure 9 *Inline transformation workflow for finding a body to inline*

The transformation also provides the dead code elimination. All method calls that lead outside of the component and that are not required are removed. Also if the component's class contains a method which is not provided by the component and which is not used by any provided method, then that method is not included in the final class. It is a side effect of merging because only provided methods are copied to the final class and the rest of methods which are called inside of provided methods are represented by their bodies which are inlined to corresponding provided methods.

Figure 10 shows an example. Class A is only class which implement component C. It shows inlining (transit method and helper field), points-to analysis example (helper field) and dead code elimination (deadCode method). Renaming of the class A to C demonstrates that C is the new class with the same name as the component and that everything is copied from the original class. Line marked A shows keeping of required names. Line marked B demonstrates result of dead code elimination. Line marked C demonstrates simple inlining. Lines marked D and E demonstrate ambitious result of points-to analysis.

```
class A {
   R required;
   S helper;
   void provided(){
     required.call();
     transit();
     helper.help();
}
   void transit(){
     deadCode();
     required.call();
   }
   void deadCode(){
System.out.println("output");
   }

   A(){ //constructor
    if (QAbst.nondet()){
      helper = new HA();
    } else {
      helper = new HB();
    }
   }
   class HA implements S {
     help(){
          required.call();
     }
   }
   class HB extends HA {
     void help(){
       required.call();
       super.help();
     }
   }}
```

```
class C {
   R required;
   void provided(){
     required.call(); //A)
     {} // B)
     required.call(); // C)
     if (QAbst.nondet()){
       required.call(); //D)
     } else {
       required.call(); //E)
       required.call();
     }
   }
}
```

Figure 10 *example of inline transformation*

3.2.5 Cleaning

The transformation is not necessary, but without it the final protocol is messed by empty blocks, loops and conditions.

Reducing Empty Branches - It removes empty blocks, loops and conditional statements. The transformation works recursively so if a block contains only empty blocks then the whole block is removed

```
switch(QAbst.nondet()){
  case A:
    do();
    break;
  case B:
    while
(QAbst.nondet){
    }
    break;
  case C:
    {}
    break;
}
```

```
switch(QAbst.nondet()){
  case A:
    do();
    break;
}
```

Reducing Nondeterministic Decisions - A conditional statement which encloses only another conditional statement is a good example of an unnecessary multiple nondeterministic decision because all conditional statements during the transformation are at this stage nondeterministic. Therefore, the two above mentioned conditional statements can be merged into one because two nondeterministic decisions about the execution of one block are the same as if this decision was made only once.

```
while(QAbst.nondet()){
  if(QAbst.nondet()){
    …;
  }
}
```

```
switch(QAbst.nondet()){
  case A:
    do();
    break;
}
```

Merging Exception Handler - The transformation also merges 'try-catch-finally' blocks and checks if there are no method calls in 'catch' blocks (after the 'inline' transformation G-AST contains only required method calls). If a method call is contained then the transformation reports an error. Otherwise the 'try' block is joined with the 'finally' block and this new block replaces the original block. Another cleaning goal is to remove unnecessary multiple nondeterministic decisions.

note: b is not required method otherwise error will be reported

```
try {
  a;
}catch (Exception e){
  b;
} finally {
  c;
}
```

```
{
  a;
  c;
}
```

3.2.6 Creation of Behavior Protocol

The final step takes the simplified G-AST prepared by previous transformations as an input and transforms it into the BP specification. It takes the prepared code and fills the provided behavior in an unrestricted way (no restriction on number of invocation and order of calling). Reactions are filled with direct translations of the transformed G-AST to the behavior protocol described in table 1. The method which is marked as a thread is allowed to do parallel execution to the rest of provided methods and also between threads.

Example demonstrates that class name is same as component name. Also that interface names in TBP is not same as interface names.

```
package example1;

public class Test {
  Require p;
public void provided(){
 while(QAbst.nondet()){
  {
    p.require();
  }
  if (true)
    Qabst.nondet();
}
}
```

```
component Test {
 types {
 }
 vars {
 }
 provisions {
   PrI {
      ((?PrI.provided())*)
   }
 }
 reactions {
  PrI.provided() {
    while (?){
      !ReI.require();
    }
    if (?){
      ReI.require()
    }
  }
 }
}
```

4 Technologies and Techniques for Transformation

4.1 Model Driven Engineering

Model driven development is a methodology which focuses on creating models or abstractions. The meta-model which is produced by model driven development is used as an input and an intermediate data model. Generated behavior protocol is not implemented in any modeling framework, but it will be possible in the future to have behavior protocol as meta-model and all transformations will become transformations between meta-models.

Development abstractions are closer to particular domain concepts rather than computing (or algorithmic) concepts [9]. Model driven development contains two main parts – Domain specific modeling language (DSML) and transformation engine.

Domain specific language is a high level language which is sometimes called meta-model. That language must be defined to match the problem specific domain. It should either be derived from an existing language or created from scratch. Language should show various facets of system, so all insiders must understand their own part of the system even if they are not programmers and do not know programming language. For example for mobile phone application it should allow high-level UI design as well as data storage model description. Language often uses graphical elements that are directly related to problem specific domain. That helps to improve the learning curve of a new developer and it also allows an easier correctness check for domain experts ([10]).

Transformation engine analyzes and synthesizes various artifacts of a model such as source code, simulation input, deployment description, various alternative model representation or meta-models. This ensures synchronization between these artifacts. This is important for easier adoption of changes and also allows domain experts, who understand only a meta-model, to check these changes. This automated transformation process is often called “correct by construction” ([10]).

The meta-model which is produced by model driven development is used as an input and an intermediate data model. Generated behavior protocol is not implemented in any modeling framework, but it will be possible in the future to have behavior protocol as meta-model and all transformations will become transformations between meta-models.

4.1.1 Eclipse Modeling Framework (EMF)

Eclipse modeling framework (EMF) is eclipse-based modeling framework and code generation facility for building applications based on structured data models. EMF generates runtime classes and adapters for utilities for viewing and editing model from model specification in XMI. The most important part is that it allows an easy interoperability between tools that are based on EMF so no

one needs to stick to one tool.

This work uses EMF meta-model as an input – G-AST is modeled in this framework. Some functionalities which are used in this work are based on EMF features with some little improvements such as deep copy or the use of EMF's inheritance hierarchy for generic functions. Another possible usage should be a transformation tool that operates on EMF and that transforms between the same G-AST models using the simplified transformations described above. Transformations between the same models are not difficult, but manual transformations, especially with a good ancestor, are better for us to understand the code (less knowledge requirements) and they are more intuitive, so for this purpose transformation tools are not used. The following two sections describe EMF transformation tools which are considered for this work.

4.1.2 Open Architecture Ware

Open architecture ware (OAW) is modular model driven architecture/ development framework for Java. It supports various tasks from parsing models, generating code from models to checking and transforming models. Not only EMF models, but also another models such as XML model, UML model or simple JavaBean model are supported.

Transformations in OAW can be done in Java or in OAW's DSL language via framework Xtext. Java transformation is done via a class which must provide a slot. This can be easily done via extending basic transformation class. Transformations can be chainloaded to final workflow which is recorded in the XML specification of actions on the target model. If a transformation is written in Java code, it is the same as if was written without this framework and needed an additional code to correctly traverse the entity tree.

Xtext framework allows to specify your own DSL language grammar. Then it automatically generates a parser for this language, a meta-model and an eclipse editor for this language. This is a real overkill for a simple transformation of model, but it contains some really interesting things. After generating your own grammar, a parser, an editor and a generator are available as eclipse plugins which increase the usability of this language. The editor provides code completion and error checking. Error checking needs defined constraints.

I think that this framework contains many interesting features and is really useful, but only for larger projects. It has a work overhead which is too big for smaller projects. This overhead is not important in larger projects as their error checking and good structuring increase the efficiency of team work.

4.1.3 QVT

QVT (query/view/transformation) is the standard for model transformation defined by the Object management group. QVT defines three different languages for transformation – Core, Relations and Operational Mappings. Relations and Core are declarative at different levels of abstraction with the mapping from Relations to Core. Relations has textual and graphical syntax and has a higher level of abstraction than Core and is used more often. Operational is an imperative language that extends both Core and Relations. Eclipse implementation of QVT is SmartQVT [11]. SmartQVT implements only operational language and uses this language for transformations.

When comparing SmartQVT to OAW, SmartQVT has a worse documentation and the general impression is that SmartQVT is not mature enough.

4.2 Points-to Analysis

The main goal of points-to analysis in Q-Abstractor is to recognize runtime type of the variable. It is done through analysis of variable assignments in a code.

Considered frameworks are WALA and Soot. Both are Java code optimization frameworks which use points-to analysis to reduce duplicated variables and to optimize registry usage (it is not physical registry but Java bytecode registry). WALA is developed by IBM and is used to research possible improvements to IBM JDK. Soot is developed at McGill university (Montreal, Quebec, Canada) and as open source it has many different contributors. For this project I chose Soot because it is better documented, there are many tools that use this framework which can be useful for this work and soot can work on higher abstraction level then Java bytecode (more can be found in the following paragraph). One of the tools that use Soot framework is Indus which does program slicing. The Indus was considered to be used to remove unneeded variables, but it does not fit the requirements and moreover, variable abstraction is an intuitive task for basic behavior protocols and it does not need any external tools.

Soot has three intermediate representations of Java bytecode: middle level Jimple, middle level Grimp and low level Baf. Jimple is a simple intermediate bytecode representation which looks like a simple Java code – instruction is in 3-address code form. It is not structured, all 'elseif', 'else', 'while' and 'for' statements are broken down into multiple statements. Goto is allowed and quite often used for this break down. The important thing is that Jimple local variables are typed. Jimple is an ideal code for general optimization such as inlining, devirtualization or copy propagation.

Grimp is similar to Jimple except that it represents statements as a tree. This is useful for decompilation or bytecode optimization. Baf is similar to bytecode but it does not contain any unnecessary information. It does not contain any encoding issues of java bytecode such as multiple

variants of the same instructions or constant pool. Baf is used to final reordering optimization. An example of Java code and all three forms is in figure 12.

For points-to analysis, the most important representation is Jimple as it provides devirtualization and points-to graph. Jimple has typed variables and since the transformation to Jimple allows to keep the original names of the variables, it is possible to identify matching local variables from the Java code.

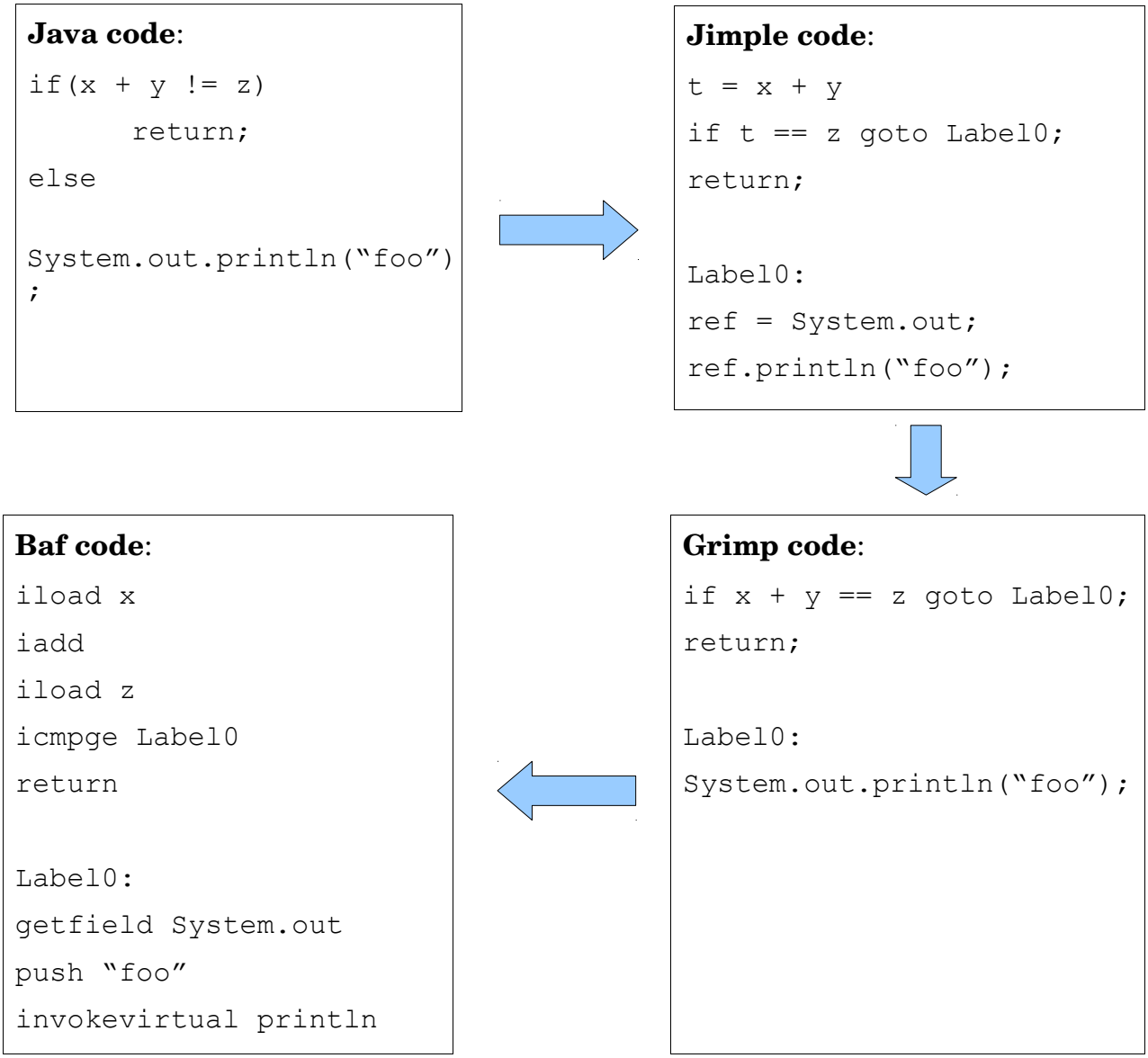


***Figure 12**: Soot inter-code example from [12]*

Spark framework which is a part of Soot is responsible for points-to analysis. Spark is also developed at McGill university by Ondřej Lhoták ([13]). It is a flexible framework for experimenting with different points-to analyses on Java code. It takes a code (in Jimple representation), a call graph and a native method simulator as the input and creates a pointer assignment graph. Afterwards, the graph is simplified by merging variables with identical points-to set. The final step is the propagation of points-to set over the edges to the whole graph. Each step

has many options which affect precision and the speed of analysis. This work uses the suggested options from Soot manual to get the most precise output, together with an acceptable performance. I used values from the manual because the goal of this work is not to compare points-to analyses, but to only use the result of points-to analysis.

Spark does not contain context-sensitive points-to analysis. Context-sensitive analysis is more precise as it takes care of the context in which a method is invoked. There are two implementations of context-sensitive analysis which are considered. The first one is paddle which is developed by Ondřej Lhoták (Spark creator). It allows to experiment with more possible algorithms. Paddle represents context facts as binary decision diagrams.

The second implementation is added to Soot by Manu Sridharan from the University of California at Berkeley ([14]). This algorithm is refinement-based context-sensitive points-to analysis and is run on top of a Spark pointer assignment graph. Context-sensitive algorithms affect three major things in a pointer assignment graph. First, they exclude all unrealized paths and analyze all calls and returns on an inter procedural basis. A heap simulator is context-sensitive, so object allocation in different calling contexts is distinguished. Finally, the most interesting thing for this work is that it constructs a context-sensitive call graph which resolves virtual calls separately for each calling context.

Both context-sensitive analyses also require JRE analyses to discover the whole context, but the analysis of the whole JRE takes an unacceptably long time. For this reason, a simple points-to analysis without context is used.

Attempt to manual configure all soot transformation and directly use its result failed. Working solution is start soot by simulation command line input and, as Soot stores its graphs in singleton which is not cleaned after command line execution, it is possible to use those graphs after all tranformation end. Soot is invoked only on target class and it is an adequately fast solution.

4.3 TBPLib

TBPLib is a library for parsing TBP specifications and for transforming these specifications to graph or tree representations. It is developed at Distributed Software Research Group at Charles University in Prague, the Czech Republic. The library is written in Java language so it can be easily used in this work. The most interesting functionality for this project is that it has data structures for TBP and it can serialize these structures in TBP files via Visitor pattern. Visitor pattern also allows everyone's own version of serialization.

Since library is ready to use and authors quickly respond to potential problems, then the library was chosen for this work. The work uses the library serialization so the result is TBP specification

without TBP variables and synchronizations. Library supports everyone's own output format, so it is possible to write pure BP output but now there is no reason for it.

4.4 Configuration

Transformation has a few optional arguments that affect transformation behavior and result. There are many configuration formats and libraries where to load and store the settings. Requirements are an easy read of the configuration (so it has to be human readable format), a simple format and an intuitive usage. Candidates for this task are XML, YAML and my own simple format.

These days XML is a common format and has an excellent support in Java. Another advantage is that it also has an integrated correctness check. It can be easily extended for future options. A file with XML is quite big and not exactly human readable. It is not intuitive to write XML configuration without XML editor with DTD support. Java can parse XML as DOM tree or via callbacks at SAX framework.

YAML is a format that uses simple text format and has three basic elements which are block, map and list. It is intuitive and has an excellent support in dynamic languages like ruby or python. It is becoming very popular, especially at ruby community. The disadvantage of YAML is that it has not such a good Java support. There exist three libraries for Java – JYaml, jvyaml and snakeyaml, JYaml is small pure Java implementation, Jvyaml is based on ruby implementation and snakeyaml is based on python implementation. In this work tested JYaml as a YAML loader.

I prefer YAML format as it fits quite well with the requirements and it is really intuitive. When I tested JYaml library, problems with typed language appeared (JYaml returns pure Object so it needs runtime checking of type) and there was also a problem with a bad documentation of this library, which broke my attempt to load a file. That is reason why JYaml and the whole YAML format are not used. XML was the second candidate. It has a good support in Java, but manual writing of configuration was not user friendly. That is why my own simple format won this format challenge. This work uses a simple format *key= value* which is intuitive, well readable and easy to use thanks to its documentation of configuration file (format allows # *comments*).

Problems during reading configuration are reported by three easy to understand exceptions. The first one is if the whole line is bad, the second one is if the key is unknown and the third one is if the value for the key is invalid. This covers all potential problems in this simple format.

5 Implementation

5.1 Language and Libraries

The program resulted from this work is written in Java. This is because Java is the analyzed language, EMF is one of Java libraries and Java is a stable and well known mature language. It uses EMF and G-AST libraries to manipulate with input. TBPlib which is mentioned in Technologies for transformation is used for output. Points-to analysis requires Soot library.

5.1.1 Code Quality Checker During Development

Code quality checker is a program that finds "problematic patterns" in a code. It uses static analysis to check the code. It helps to find bugs, performance problems or bad style code. It helps to improve code quality, especially in a phase when there is not enough data for full testing. It should be sufficiently fast and verbose enough about the found problems.

I chose FindBugs because I knew it from my other projects and it provided good results. FindBugs is an open-source program which was started at the University of Maryland. Now it is financed by such companies as Google, Sun Microsystems or IBM (via Eclipse foundation).

FindBugs helps to improve code quality. It is enough for us to specify directory where source codes are stored and FindBugs automatically analyzes the whole program.

5.2 Architecture of Solution

Q-Abstractor uses the Transformer object to organize transformation. It gives each transformation a G-AST tree on which each transformation works along with information about components in Metadata Extractor. Figure 6 shows the way how a G-AST tree goes through the program. TBP creator does not change the G-AST tree – only produces a TBP tree.

5.3 Transformation

All transformations have the same ancestor which implements basic operations. The aim of the first operation is to predefine methods to traverse tree. Default action for each node is no action, it only calls nodes under the current node. Current transformation overwrites processing nodes if they need to make any action and if the transformation also wants to process child nodes it calls an ancestor method.

The second operation replaces the node with a new node and updates information about this change to parent node. It uses information about parent statement in the node and replaces it via standard collection method. Replacing can also be used to remove a statement. If a new statement is

set to 'null', then an old statement is only removed.

The third operation is deep copy of the statement which is implemented in G-AST copier which is described in the following section.

5.3.1 G-AST copier

GASTCopier is a wrapper around Copier from EMF Ecore utils. The original copier is used to do deep copy of EMF structure. It also copies references and its usage is quite simple. It was found during testing that it does not copy accessed target for functions and variables, which is important to recognize which function or variable was used. This was solved by the wrapper which extended functionality and which manually copied accessed targets during copying of Variable or Function. The usage of the wrapper allows an easy handling of similar problems or little changes of behavior which is then automatically used in each transformation.

5.3.2 Modularity

Variable points-to analysis - Each implementation of points-to analysis which implements 'PointsTo' interface can be used in the transformation. Now it is possible to use Soot implementation and dummy implementation. Soot implementation is an ancestor of dummy implementation as there are some tasks that Soot does not solve and then dummy solution is used for these tasks.

Component definition - Component can be defined in various ways. All implementations must return component representation which implements `MetadataExtractor` interface. It can be a direct implementation of `MetadataExtractor` which has statically defined components. Predefined way is component specification in a file. Another possible solution can be to dig information from SAMM Component abstraction when the required information about SAMM implementation is known. Also, a component which is marked by annotation can be used when G-AST annotations are fixed.

Automatic addition of interfaces to component - Interfaces which are implemented by any of the component classes should be automatically added to the component. It decreases execution speed as the investigated space has increased. But it prevents problems when a method is called on an interface and the interface is ignored because it is outside of the component, but during runtime a variable with that type could have an assigned component class which contains an important call.

Return replacement - It is mentioned in the section about Jump Replacer that this transformation is optional. It enables automatic enclosing of method remaining statements in conditional execution. The transformation decreases the readability of protocol, but the protocol is correct. Without this transformation, some work-flow could be skipped, but the final protocol is

cleaner and easier to check.

5.3.3 Assumptions of analyzed code

The table describes all limitations which Q-Abstractor cannot handle and how to work around it.

Problem	*Reason*	*Workaround*
Recursive call which contains required call	Inline places method body instead of its call. So it creates an end-less loop, which Q-Abstrator reports if find in recurse required call otherwise call is ignored.	Replace recursion with stack and loop.
Invoking method on an object which is the result of another method e.g. *a.get().do();*	Points-to analysis is not prepared for this case	Replace it with saving results to local variables e.g. *C c = a.get(); c.do();*
Required call in catch block of an exception	Component system doesn't expect this case	Work-around doesn't exist.
Fall-through switch blocks	TBP doesn't support it	Change code

6 Future work

Q-Abstractor needs adaptation to changes at other projects of Q-Impress and also adaptation to expected result. There is still place to improvement of analysis precise and big one is consider data which affect analysis.

6.1 Transformation with Data

In this section one will find a description of possible extension of transformation that can be

```
switch a:
{
  case POSITIVE:
    switch b:
    {
      case POSITIVE:
        c = POSITIVE;
      case ZERO:
        c = POSITIVE;
      case NEGATIVE:
        c = POSITIVE | ZERO | NEGATIVE;
    }
  case ZERO:
    c = b;
  case NEGATIVE:
    switch b:
    {
      case POSITIVE:
        c = POSITIVE | ZERO | NEGATIVE;
      case ZERO:
        c = NEGATIVE;
      case NEGATIVE:
        c = NEGATIVE;
    }
}
```

***Figure 15**: Transformation of add operation on integer type: c = a +b;*

used to generate threaded behavior protocols (TBP) instead of BP. TBP contains more detailed

information about component behavior so it is harder to generate it.

6.1.1 State Variables

The first extension are state variables. State variables are described in the section about Thread behavior protocols. In short, these are the variables which affect control-flow of code execution. It is often used where nondeterministic expression is used in behavior protocols. It is more precise when defining possible execution order. The first goal is to identify these variables in the real code. Simple solution is the marking of these variables in the code by the user (comment, special name etc.). Automatic recognition by automatic identification is a more sophisticated solution. Automatic identification finds any variables on which invocation of required method depends. These are especially the variables used in condition statements and loops. So any variables in `switch`, `if` or in break condition in loop statements whose bodies contain important function calls are marked as state variables. This recognition is transitive, so if a variable affects another variable which is a state variable then the first variable is also a state variable. Figure 16 shows example of transitive recognition.

```
public void abort () {
   if (file_open)
     a.close; //required call
}

public void clean(){
   if (close_all){
   …;
   file_open = false;
   }
}
```

Figure 16: *transitive state variable recognition. Because 'a.close' is a required call, then 'file_open' is a state variable. Because 'close_all' affects 'file_open' then also 'close_all' is a state variable.*

State variables require a new transformation which transforms variables from the source language to TBP types. TBP allows only enumeration types. It requires a structure that maps constants and invocation of methods on that variables to finite state machine. It looks as if Kripke structure is sufficient for this task. Kripke structure is a type of nondeterministic finite state machine which holds information in states. In our case, each state represents an enumeration value of the

represented type and each transition relation represents a method invoked on a variable. The initial state of the machine is an uninitialized state (same as null in Java). To assign a constant is also an operation which converts the constant to enumeration value and sets this value to the machine. This also works for more complex structures which are moved from their uninitialized state via their constructor. This is done by analyzing the constructors and how they affect internal data of a structure. Any operation with a variable can be defined as a transition between states of the machine and this leads to a very flexible tool which can define any possible method or build-in function. Methods can have arguments and in this case the transition relation can fix the argument to one enumerable value (as all variables which affect a state variable must be state variables and so they also are enumerable) e.g. to separate plus operation to plus state A and plus state B. How to get this structure is still open to question. Some parts may be generated automatically and some parts must be defined by the user.

The current state of a variable cannot often be statically determined. This can lead to a very complex set of conditions which set a new state. In figure 15 there an example of that complexity for generic assign result of add operation for integer type with simple states for negative, positive and zero numbers. Complexity increases with the raising number of parameters and different states of these variables. This can be specified for the worst case as $\prod a_n$, where a with index n is the count of states of argument n. In short, the result in this case is the full count of transitions in Kripke structure.

TBP also adds arguments to method calls and return value. It does not necessarily mean that all arguments from Java code must be included to TBP. It is the same as for state variables where only the variables which affect (both directly or indirectly) execution of required methods are important. So method signature and calling should also be affected by this transformation.

This new transformation mainly affects expressions and it can be merged with 'remove non-representable' transformation as the latter transformation loses a lot of its functionality. Also, if 'remove non-representable' transformation is not merged, it must be changed to let operators on the variables and it is not easy to extract a method call without affecting that expression.

State variables often are fields of class and inline phase must reflect it. Inline transformation needs to merge these fields and rename them if necessary. Another task of this transformation is to identify shared state variables. Shared state variables are fields which point to the same data and are marked as state variables. Shared state variables must be identified and represented as one field in the merged class. The reason is that when state variables share they state and one of them changed its state, the state of the remaining variables also change and it should affect the behavior of another part of the component. Points-to analysis provides the result of possible targets for these variables

but there is still a problem when the analysis cannot determine the result. Then the user must choose if shared stated variables with an undetermined result should merged into one state variable or if they should be let as separated state variables.

6.1.2 Threads and synchronization

TBP supports threads and basic synchronization. Different behavior of threads in TBP and in Java constitute the first challenge for the transformation. TBP threads cannot be dynamically created – the number of threads stays constant during computation. Java threads can be created dynamically – nobody knows before execution how many threads will occur. The analysis of the behavior of a thread stays the same as for another methods.

The whole method synchronization can be simply converted to TBP synchronized blocks. Transformation of various Java synchronization primitives to Mutex is more difficult – Mutex is only a synchronization primitive which supports TBP. This can produce a quite unintuitive code for some primitives such as semaphore or barrier. Also, all method calls must be tracked to find out if they do not use any synchronization and if they properly match to corresponding primitives. All synchronization primitives are marked as important variables and are added to protocol variables.

7 Related work

The section describes a few tools which have a similar goal and it also describes how they reach it. Each tool description contains its goal, the way how it tries to reach it and a comparison to Q-Abstractor.

7.1 Xg++ and Murφ [15]

The goal of Xg++ is to extract high-level specification of C code to Murφ language. Xg++ is a compiler which allows writing of domain specific extensions. Murφ is a pascal like language which contains explicit states and a checker of those states. Xg++ uses slicing and rewriting rules on abstract syntax tree to accomplish its goal.

If a user wants to use Xg++ then he must not only have a source code, but he also has to add which variables are state variables and he must also define his own transformation rules. His transformation rules help to improve the transformation, because they add domain specific knowledge. If a user wants to use Murφ, he not only has to have the result of Xg++, but he also needs to define initial states, correctness properties and hardware model. Correctness properties contain invariants and asserts of model which are checked on errors. Hardware model is required to understand assembler code in C code. Thus, if another function is invoked in an assembler, then hardware model helps to understand it.

If I compare Q-Abstractor and Xg++ with Murφ then Q-Abstractor does not extract state variables, does not check correctness itself and does not allow an easy way to define its own transformation. On the other hand, Q-Abstractor is more automatic, requires almost no user input, it is easier to use and the resulting BP is more readable.

7.2 Extraction Based on Finite State Machine [16]

The goal of this work is to model interface of class as multiple finite state machines. States represent some methods of class and transitions are the allowed consecutive methods. Methods are often the methods which represent a certain implemented interface or which access a certain field. This work uses static analysis to determine illegal call sequence, dynamic instrumentation techniques to extract model from execution runs and dynamic model checker that ensures correct call order.

When compared to Q-Abstractor, its focus is to determine the allowed consecutive methods so they have a different goal. But they can be used together. Q-Abstractor could create reactions and this tool could create provisions. But there is a restriction for this tool. The restriction is that during dynamic extraction there are bounces at methods marked as synchronized, so it slightly changes the

behavior of the application.

7.3 Java Interface Synthesis Tool (JIST) [17]

JIST goal is to automatically generate specification of correct method call sequences of Java class from its implementation. As a first step it constructs a symbolic representation of finite state-transition system using predicate abstraction. Then it constructs an interface that corresponds to solving partial-information two-player game (algorithm from the theory of games). Algorithms for learning finite state machines are used for this computation hard problem and symbolic model checking is used for branching. The implementation contains four steps. The first step is to convert from Java to Jimple which is soot internal format. Then Jimple is converted to Boolean Jimple using predicate abstraction and a set of predicates. Boolean Jimple is then converted to boolean model in NuSMV (symbolic model checker). The final step is to create an interface from NuSMV. The interface can range from the most permissive interface to a maximum save interface.

When compared to Q-Abstractor, JIST is more data oriented. JIST uses advantage algorithms to provide good results. Due to its complexity JIST allows only a subset of Java. Moreover, its performance is not ready to use on-fly during developing, but it is adequate to use e.g. on build server. JIST acts on a single class so it does not need to solve points-to analysis and other issues connected with sets of classes such as inlining.

8 Conclusion

Q-Abstractor successfully creates behavior protocol if code does not contain complies assumptions. Detected violence of assumptions is properly reported. It demonstrates that automatic behavior caption is possible. Q-Abstractor has partial integration to Q-Impress chain of tools. It provides working base for possible future extensions. The code is well documented and prepared to future changes.

This work increased my experience with the development of a software which must cooperate with other developed software systems. I realized how important is defensive programming techniques to developed libraries as it can contain bugs and often change its API. I realized how important deployment (installation and way how to run it) was even in early stages for catching problems. Deployment also should contain basic documentation how to run an application and how to avoid known problems. I also realized that many interesting research works do not reach final state where the work is ready for practical usage and that the works lack documentation for practical usage (however theoretic documentation is often very good). For example, my attempts to run soot as a library took more than a month and afterwards the solution is quite simple, just not documented.

The work has had to deal with some difficulties. The first one is that a work with the same goal (to convert from abstract syntax tree to reactions in behavior protocol) does not exist. So I met with many dead ends which would otherwise have been mentioned in the documentation of similar projects and I could not take inspiration from the ways how similar projects tackled their problems. It led to the fact that this work required my own way to implement such functionality instead of choosing the most fitting one from already existing solutions.

The biggest difficulty is that G-AST is developed at the same time as this work. This leads to problems with changes in api, providing testing data (quite a long time is needed to program just against an interface because no data is available). Deployment of G-AST (especially to convert sissy result to G-AST model) in early stages is difficult and takes a serious amount of time. And some parts of G-AST are available only in later stages of development which leads to big changes in code as interfaces are changed to fit implementation.

Even if the final code is quite small, it is a product of many refactorings and cleanings. During its development, it contained many workarounds for broken or not yet implemented things in G-AST and also attempts to find better implementation. But as G-AST started working better and dead ends implementation were found, then the code was getting smaller. In addition to this, using libraries and java core functionality helped to keep the code small, clean and problem oriented.

What I would change if I was at start now is that I would wait a year until G-AST was more stable and then I would start programming, which would mean avoiding a lot of unnecessary work.

Bibliography

[1] Tomáš Poch, František Plášil. Extracting Behavior Specification of Componens in Legacy Applications. Charles University in Prague - 2009.

[2] Petr Hnětynka. Component Model for Unified Deployment of Distributed Component-based Software. Charles University, Faculty of Mathematics and Physics, Dept. of Software Engineering - 2004.

[3] Hnetynka, P., Plasil, F., Bures, T., Mencl, V., Kapova, L.. SOFA 2.0 metamodel. Dep. of SW Engineering - 2005.

[4] Kofron, J., Poch, T., Sery, O.. TBP: Code-Oriented Component Behavior Specification. IEEE - 2009.

[5] Plasil, F., Visnovsky, S., Besta, M.. Behavior Protocols. Dep. of SW Engineering, Charles University, Prague - 2000.

[6] Pavel Parizek. Extraction of Component-Environment Interaction Model using State Space Traversal. Charles University in Prague - 2009.

[7] Steffen Becker, Lubomír Bulej, Tomáš Bureš, Petr Hnětynka, Lucia Kapová, Jan Kofroň, Heiko Koziolek, Johan Kraft, Raffaella Mirandola, Johannes Stammel, Giordano Tamburrelli,Mircea Trifu. Project Deliverable D2.1Service Architecture Meta-Model (SAMM). - 2008.

[8] SISSy - http://sissy.fzi.de.

[9] model driven engineering - http://en.wikipedia.org/wiki/Model_driven_development.

[10] Douglas C. Schmidt. Model-Driven Engineering. IEEE ComputerVanderbilt University - 2006.

[11] France Telecom. SmartQVT documentation. - 2007.

[12] Raja Vallee-Rai, Patrick Lam, Patrice Pominville, Feng Qian, and Laurie Hendren. Soot - a Java Bytecode Optimization and Annotation Framework. - 2000.

[13] Ondřej Lhoták. A flexible points-to analysis framework for Java. - 2003.

[14] Manu Sridharan, Rastislav Bodík. Refinement-Based Context-Sensitive Points-To Analysis for Java. PLDI - 2006.

[15] David Lie, Andy Chou, Dawson Engler, David L. Dill. A Simple Method for Extracting Models from Protocol. Computer Systems Laboratory, Stanford university - 2001.

[16] John Whaley, Michael C. Martin, Monica S. Lam. Automatic Extraction of Object-Oriented Component Interfaces. Computer Systems Laboratory, Stanford University - 2002.

[17] Rajeev Alur, Pavol Černý, P. Madhusudan, Wonhong Nam. Synthesis of Interface Specifications for Java Classes. Department of Computer and Information Science, University of Pennsylvania - 2005.

9 Appendixes

A) Installation, User guide

9.1 Installation

Q-Abstractor installation is simple. Just copy the directory and the tool is ready to use. There is a predefined configuration file *transformation.conf* and a few runnable examples.

9.2 Compilation

Q-Abstractor uses Ant building system. For compilation is enough to type `ant`. To generate final jar file type `ant dist`.

9.3 Dependencies

Q-Abstractor requires G-AST libraries, EMF core libraries, Soot library and TBP library. All dependencies is in *lib* directory.

9.4 User Guide

9.4.1 Requirements

Q-Abstractor needs Java Runtime Environment at least 1.6. It supports all platform where JRE is supported.

Q-Abstractor requires G-AST tree serialized in XML and component description to generate TBP specification.

9.4.2 Examples

Examples provides predefined inputs. Q-Abstractor contains three examples to demonstrate its abilities. It is in the directory *tests*. Each test has its own directory with several sub-directories. Java source code can be found in sub-directory *src*, generated G-AST tree and component description in the tool's own format can be found in *input* sub-directory. There is an executable file *runtest.bat* which runs the tool on *input* sub-directory. The executable file can be run on Windows and also on Unix via *sh runtest.bat*. The result is created in the directory from which the tool was executed. It creates tbp files which have the same name as the component which they contain.

Inlining virtual methods test – the example demonstrates inlining of methods

Session manager – example component which has thread

9.4.3 Analyzing own sources

If you want to run Q-Abstractor on your sources, then you must create own inputs.

Component Specification

Component specification contain for sections. Values are specified after colon and one per line. Sections are *component* for component name, *classes* to specify which classes implement the component, *required* to specify required interfaces and *provided* to specify provided methods. Component section expects the name of the component to be specified on one line. Classes keyword expects fully qualified class name. Required and provided sections expect fully qualified path to interface (required) or method (provided) then symbol → (minus and greater then) and the name of the interface which defines that interface or method. An example configuration file can be seen in figure 14 and in each example.

```
component:
    SessionManager
classes:
    sessionmanager.SessionManager
required:
    sessionmanager.BusinessLogic -> BusinessLogic
    sessionmanager.Hash -> Hash
    sessionmanager.Log -> Log
    sessionmanager.GUINotification -> Notification
    sessionmanager.Database -> DB
provided:
    sessionmanager.SessionManager.createSession -> Session
    sessionmanager.SessionManager.invokeCommand -> Session
```

Figure 14 *– Example component specification file*

Create G-AST

For detail informations see SiSSy documentation as it is still changing (also if G-AST metamodel is changed after the release of this thesis then the tool could stop working and Q-Abstractor will need a modification).

At first install Eclipse with its modeling framework bundled-in. Then add software sites:

http://q-impress.ow2.org/

From that sites install whole content. Create project with existing sources or use existing one if it is already created.

Create new run configuration. Choose type Launch SiSSy. Check *Run Extraction*, specify input path and check *export model to database* on SISSy tab. For simple run choose Derby db and path to databse path on database setting tab. On third tab check all checkboxs and specify three valid paths to file. Merged file is requested G-AST tree. Then click on run and file is generated.

Configuration

It is possible to modify Q-Abstractor behavior by specifying configuration file. The file now contains three options. The first option is points-to analysis implementation. The only supported value is dummy because there is missing information about the class path in soot implementation from SiSSY (but will be added). The second one is automatic inclusion of interface to component. It helps to inline calls which are invoked on the interface which is not specified as a part of component but some of its implementation is in the component. Default value is true. The third one is enclosing the rest of the method which contains the return to conditional execution. Default is true, which creates correct behavior protocol, but that protocol can be sometimes confusing so it can be disabled.

9.5 Integration to Eclipse

The whole Q-Impress project will be integrated to eclipse platform and provide plugins which will allow an easy usage of all tools. It is not yet fully done. Q-Abstractor now provides a menu item which will be in the final menu for analyzing tools. It will be available after the release of this work so it cannot be placed on the CD.

B) Content of CD

doc/ – contains documentation to project

doc/api – generated javadoc for Q-Abstractor

doc/image – source of images in the thesis

dist/ - generated jar files which can be copied and it is ready to use

lib/ - libraries needed to run and compile Q-Abstractor

src/ - sources of Q-Abstractor

tests/ - contains examples

tools/ - contains code quality checker used during development

Q-Abstractor-plugin/ – contains eclipse plugin which provides menu item with Q-Abstractor

./build.xml – ant configuration for Q-Abstractor

./transformation.conf – commented configuration file for Q-Abstractor

./README – short notes how to compile and run Q-Abstractor